### CPSC 213

**Introduction to Computer Systems**

**Unit 2c**

**Synchronization**

### Implementing Atomic Exchange

- **Can not be implemented just by CPU**
  - must synchronize across multiple CPUs
  - accessing the same memory location at the same time

- **Implemented by Memory Bus**
  - memory bus synchronizes every CPUs access to memory
  - the two parts of the exchange (read + write) are coupled on bus
  - bus ensures that no other memory transaction can intervene
  - this instruction is much slower, higher overhead than normal read or write

### Spinlock

- **A Spinlock is**
  - a lock where waiter spins on looping memory reads until lock is acquired
  - also called “busy waiting” lock

- **Simple implementation using Atomic Exchange**
  - spin on atomic memory operation
  - that attempts to acquire lock while
  - atomically reading its old value

### Implementing Spinlocks Efficiently

- **Spin first on fast normal read, then try slow atomic exchange**
  - when lock appears free use exchange to try to grab it
  - if exchange fails then go back to normal read

### Blocking Locks

- **If a thread may wait a long time**
  - it should block so that other threads can run
  - it will then unblock when it becomes runnable (lock available or event notification)

- **Blocking locks for mutual exclusion**
  - if lock is held, lockers puts itself on waiter queue and blocks
  - when lock is unlocked, unlocker restarts one thread on waiter queue

- **Implementing blocking locks presents a problem**
  - data structure includes a waiter queue and a few other things
  - data structure is shared by multiple threads; lock operations are arbitrary interleaved
  - mutual exclusion can be provided by blocking locks (they aren't implemented yet)

  - and so, we need to use spinlocks to implement blocking locks (this gets tricky)
Implementing a Blocking Lock

‣ Spinlock guard
  • on for critical sections
  • off before thread blocks

struct ...  if (waiter_thread) {
    waiter_thread->state = TS_RUNNABLE;
    ready_queue_enqueue (waiter_thread);
  }
}

Monitor automatically exited before block on wait
• before waiter blocks, it exits monitor to allow other threads to enter

void refill (int n) {
  monitor {
    for (int i=0; i<n; i++) {
      glasses++;
      notify;
    }}}

Confusion about when spinlocks needed
• must turn on to guard access to shared variables
• must turn off before finishing or blocking

Confusion about loop function
• busywait
  • only inside spinlock
• thread blocked inside loop body, not busywaiting
  • when finishing, re-check for desired condition: lock available?

Condition Variables
• Mechanism to transfer control back and forth between threads
  • uses monitors: CV can only be accessed when monitor lock is held

Primitive
• wait blocks until a subsequent signal operation on the variable
  • notify unblocks one waiter, continues to hold monitor
  • notify_all unblocks all waiters (broadcasts), continues to hold monitor

Each CV associated with a monitor
• Multiple CVs can be associated with same monitor
  • independent conditions, but guarded by same mutex lock

Wait and Notify Semantics
• Programs can have multiple independent monitors
  • so a monitor implemented as a "variable" [a struct really]

Monitors and Condition Variables
• Mutual exclusion plus inter-thread synchronization
  • introduced by Tony Hoare and Per Brinch Hansen circa 1974
  • abstraction by synchronization primitives in Java etc.

Monitor
• monitor guarantees mutual exclusion with blocking locks
  • primitives are enter (lock) and exit (unlock)

Condition Variable
• allows threads to synchronize with each other (provides control transfer between threads)
  • wait blocks until a subsequent signal operation on the variable
    • notify unblocks waiter, but continues to hold monitor (Hansen)
    • notify_all unblocks all waiters and continues to hold monitor
  • can only be accessed from inside of a monitor (i.e., with monitor lock held)

Monitor vs Condition Variable
• Battle of the Abstractions
  • monitors: mutual exclusion with blocking locks
  • condition variables: lightweight synchronization

Drinking Beer Example
• Beer pitcher is shared data structure with these operations
  • pour from pitcher into glass
  • refill pitcher

Implementation goal
• synchronize access to the shared pitcher
  • pouring from an empty pitcher requires waiting for it to be filled
  • filling pitcher releases waiters

Using Condition Variables for Disk Read
• Blocking read
  • call async read as before
  • but now block on condition variable that is given to completion routine

Read completion
• called by disk ISR as before
  • but now notify the condition variable, restarting the blocking read call

Shared Queue Example
• Unsynchronized Code
  • queue is not initialized
  • thread calls into the queue
  • queuerocessor
  • deadlock
Adding Mutual Exclusion

void enqueue (uthread_queue_t* queue, uthread_t* thread) {
  uthread_monitor_enter (&queue->monitor);
  if (queue->head == 0)
    queue->head = thread;
  else
    queue->tail=0;
  queue->tail->next = thread;
  queue->tail = thread;
  uthread_monitor_exit (&queue->monitor);
  return thread;
}

Some Questions About Example

Why is does dequeue have a while loop to check for non-empty?

Why must condition variable be associated with specific monitor?

Why can we use condition variable outside of monitor?

Implementing Condition Variables

• Some key observations
  • wait, notify and notify_all are called while monitor is held
  • the monitor must be held when they return
  • wait must release monitor before locking and re-acquire before returning

• Implementation
  • in the lab
    • look carefully at the implementations of monitor enter and exit
  • understand how these are similar to wait and notify
  • use this code as a guide
  • you also have the code for semaphores, which you might also find helpful
  • Google “semaphores condition variables birrell”

Synchronization in Java

• Policy question
  • monitor state is head-for-reading
  • thread A calls monitor_enter() and blocks waiting for monitor to be free
  • thread B calls monitor_enter_read_onlyly(); what do we do?

• Disallowing new readers while writer is waiting
  • is the fair thing to do?
  • but, a group of readers can access monitor concurrently

• Reader-Writer Monitors
  • monitor state is one of
    • head-for-reading
    • head-for-writing
    • locked
  • wait for monitor to be free or hold-for-reading, then sets is state to head
  • wait for monitor to be free or hold-for-reading, then sets is state to head
  • normally other provide a fair implementation
  • or allow programmer to choose (that’s what Java does)

• Why can’t we use condition variable outside of monitor?
  • this is actually required sometimes... can you think where?

• Why is does dequeue have a while loop to check for non-empty?
  • this is actually required sometimes... can you think where?

• What if you reversed order of V and P?
  • you need to keep track of the number of threads waiting for the warm beer
  • then call V that number of times
  • this is actually quite tricky

Using Semaphores

• Explicit locking not required when using semaphores since atomicty built in
  • set initial value of empty when creating it

• Using Semaphores to store glasses
  • in a system of 1 parent thread and N children threads
  • all threads must arrive at barrier before any can continue
  • barriers: all threads must arrive at barrier before any can continue

Other ways to use Semaphores

• Asynchronous Operations
  • create outstanding_request semaphore
  • async_read: P(outstanding_request)
  • completion interrupt: V(outstanding_request)

• Rendezvous
  • two threads wait for other before continuing
  • create a semaphore for each thread initialized to 0

• Policy question
  • monitor state is head-for-reading
  • thread A calls monitor_enter() and blocks waiting for monitor to be free
  • thread B calls monitor_enter_read_onlyly(); what do we do?

• Disallowing new readers while writer is waiting
  • is the fair thing to do?
  • but, a group of readers can access monitor concurrently

• Reader-Writer Monitors
  • monitor state is one of
    • head-for-reading
    • head-for-writing
    • locked
  • wait for monitor to be free or hold-for-reading, then sets is state to head
  • wait for monitor to be free or hold-for-reading, then sets is state to head
  • normally other provide a fair implementation
  • or allow programmer to choose (that’s what Java does)

• Why can’t we use condition variable outside of monitor?
  • this is actually required sometimes... can you think where?

• Why is does dequeue have a while loop to check for non-empty?
  • this is actually required sometimes... can you think where?

• What if you reversed order of V and P?
  • you need to keep track of the number of threads waiting for the warm beer
  • then call V that number of times
  • this is actually quite tricky
We have already seen this with semaphores

Consider a system with two monitors, a and b

What about now?

Recursive Monitor Entry

What should we do for a program like this

Here is implementation of lock, is this okay?

Avoiding Deadlock

Don't use multiple threads
  • you'll have many idle CPU cores and write asynchronous code

Don't use shared variables
  • if threads don't access shared data, no need for synchronization

Use only one lock at a time
  • deadlock is not possible, unless thread forgets to unlock

Organize locks into precedence hierarchy
  • each lock is assigned a unique precedence number

Before thread X acquires a lock, it must hold all higher precedence locks
  • ensures that any thread holding i can not be waiting for i

Detect and destroy
  • if you can't avoid deadlock, detect when it has occurred
  • break deadlock by terminating threads (e.g., sending them an exception)